

# BASEL (Buffering Architecture SpEcification Language)

Kirill Kogan<sup>1</sup>, Danushka Menikkumbura<sup>2</sup>, Gustavo Petri<sup>3</sup>, Youngtae Noh<sup>4</sup>, Sergey Nikolenko<sup>5</sup>, and Patrick Eugster<sup>2,6</sup>

<sup>1</sup>IMDEA Networks Institute

<sup>2</sup>Purdue University

<sup>3</sup>Université Paris Diderot - Paris 7

<sup>4</sup>Inha University

<sup>5</sup>Steklov Mathematical Institute

<sup>6</sup>TU Darmstadt

October 15, 2015

## Abstract

Buffering architectures and policies for their efficient management constitute one of the core ingredients of a network architecture. In this work we introduce a new specification language, BASEL, that allows to express virtual buffering architectures and management policies representing a variety of economic models. BASEL does not require the user to implement policies in a high-level language; rather, the entire buffering architecture and its policy are reduced to several comparators and simple functions. We show examples of buffering architectures in BASEL and demonstrate empirically the impact of various settings on performance.

## 1 Introduction

Design and management of a *buffering architecture* are key elements in meeting network design challenges since they directly impact the performance and cost of each network element. Application-induced traffic bursts can create an imbalance between incoming and outgoing packet rates to a given port, so packets must be queued in the network element. The available queue size on a port determines the port’s ability to hold packets until the egress port can emit them. Packets are dropped when buffer resources are congested, resulting in poor performance. The allocation and availability of buffer resources to ports is determined not only by the buffer’s size but also by the buffering architecture of the network element. Overprovisioning in terms of buffer capacity at each network node to absorb bursty behavior is not viable, as networks cannot have unlimited resources; on the contrary, data centers can only scale out as fast as the effective per-port cost and power consumption. These factors are, in turn, defined by the chosen buffering architecture.

Objectives beyond *fairness*, and the incorporation of additional traffic properties, lead to new challenges in the implementation and performance for traditional switching architectures [9, 16, 17, 19]. This calls for novel abstractions that enable the definition of buffering architectures and management policies that can be deployed on real network elements. Designing such abstractions however is non-trivial, as they must satisfy a number of possibly conflicting requirements: (1) EXPRESSIVITY: expressible policies should cover a large majority of buffering architectures representing common economical models of existing and future networks; (2) PERFORMANCE: the implementations of policies should be efficient on “virtual switches”, that is with various resolutions from a single network element to the whole network (e.g., an interconnect for geographically distributed data centers [17, 19]); (3) SIMPLICITY: policies for different economical models should be expressible concisely, i.e., the code size and the organization of the code should be immediately reflective of the intent of the buffering infrastructure architect.

In this work we propose the *Buffering Architecture SpEcification Language* (BASEL), a flexible way to meet these requirements and define buffering architectures and management policies that can be deployed on real network elements.

## 2 Related Work

The active networks [34] approach to programmable networks is to execute code contained within packets on the switches. However, we argue that running arbitrary code can hamper switch performance. Frenetic [15], Pyretic [25], and Maple [35], among others, have proposed abstractions to express management policies in packet networks. These approaches focus on abstractions for flexible *classifiers*, and do not try to manage buffering architectures. Other systems [13, 30, 33] allow for setting a *predefined set* of parameters for buffer management, which intrinsically limits expressivity. Another line of research abstracts the representation of the southbound API (e.g., OpenFlow) in the data plane [10, 22, 32], while languages such as P4 [10] are very successful in representing hierarchical tuple matching with action sets, we believe that they are less suitable to express buffer management policies. Usually queueing modules are physically separated from the *packet processing engines* (PPEs) implementing tuple-based classification [2, 3]. This makes it difficult to access the current state of the queueing module. Conversely, adding classification to a queueing module can significantly increase the implementation cost, let alone add performance overheads.<sup>1</sup> We believe that buffer management policies should be built on different principles. The closest work to BASEL is [31] which introduces a set of primitives to define buffer management policies. While BASEL only requires a set of comparators and conditions, [31] needs a specification of the *whole algorithm* for the management policy based its APIs, hence hampering simplicity. In addition, [31] provides no clear separation from the classification module and the interface to express desired objectives. BASEL aims to overcome these limitations.

## 3 Buffering Architecture Design

The majority of buffering architectures can be defined with only two types of objects: *ports*, and *queues* assigned to ports; in the *buffered crossbar* architecture [20], cross-points can also be represented as ports. An *admission control policy* for a queue determines which packets are admitted or dropped [12, 14, 27]. A *scheduling policy* for a port selects a queue whose *head-of-line* (HOL) packet will be processed next [11, 23]; in each queue, the HOL packet (and thus the processing order) is defined by a *processing policy*.

In some cases, e.g., in a *shared memory* switch [7], several queues share the same *buffer* space, and admission control can routinely query the state of several queues; for instance, the Longest-Queue-Drop (LQD) policy drops packets from the longest queue in case of congestion [7]. To cover these buffering architectures, we introduce one more object type, a *buffer*, and an additional admission control policy to resolve congestions at the buffer level. In a nutshell, to define a specific buffering architecture and its management policy one creates instances of ports, queues, and buffers, and specifies relations among them; admission control, processing, and scheduling policies are attached to the corresponding instances. The purpose of BASEL is to enable concise specification of buffering architectures and management policies.

## 4 BASEL Specification Language

The abstractions introduced by BASEL reconcile simplicity and expressivity. BASEL is a specification language, hence all programming decisions have to be made at the declaration of the different entities, i.e., entities are static. In the following we shall present the different entities manipulated by BASEL by means of a simple declaration of data structures (with no types). For each entity, we will define its properties, some of which are primitives of the domain (e.g., the size of a packet), and others which have to be set up by the programmer. Similarly, some properties are constant (e.g., maximum buffer capacity), whereas others are dynamic (e.g., the occupancy of a buffer). We show the characteristics of each property as comments. We denote by `r` properties that are read-only, and by `rw` properties that can also be updated by the user. Properties that are constant throughout the lifetime of the policy are marked with `cons`, and with `dyn` we mark properties that can change dynamically. Finally, functions are annotated with their return type (e.g., `bool fun`).

### 4.1 Packets

In BASEL, the notion of a packet is *primitive*, meaning that the user cannot modify or extend packets; packet fields can be used to implement policies. Since a virtual switch can be defined with any resolution, from a real switch (or a part of it) to the entire network, and can represent the buffering architecture of different services, the notion of a packet is *virtual* and is not related to specific traffic types. To be independent of traffic types and switch resolution, and to have a clear separation from the

<sup>1</sup>A single TCAM access on Cisco C12000 linecard [1] requires 11 clock cycles, where the whole IP packet processing in one pipeline stage should be completed in 31 cycles to guarantee the required line rate; running the second consecutive lookup that is based on dynamically changed values can degrade performance by up to 40%.

---

```

Packet {
  size      // size in bytes      [r, cons]
  value     // virtual value      [r, cons]
  processing // # of cycles        [r, dyn]
  arrival   // arrival time       [r, cons]
  slack     // offset in time     [r, cons]
  queue     // target queue id    [r, cons]
}

```

---

Figure 1: BASEL’s packet primitive

---

```

Queue {
  // primitive properties
  currSize      // current size      [r, dyn]
  getHOL()      // head-of-line pkt [packet fun]

  // user-specified at declaration
  size          // size in bytes      [r, cons]
  buffer        // buffer where allocated [r, cons]
  procPrio(p1,p2) // processing prio comp. [bool fun]
  admPrio(p1,p2) // pushOut prio comp.   [bool fun]
  congestion()   // congestion predicate [bool fun]
  postAdmAct()   // {MARK,NOTIFY,..}     [action fun]
  weightAdm      // priority for admission [rw, dyn]
  weightSched    // priority for scheduling [rw, dyn]
}

```

---

Figure 2: BASEL’s queue primitive

classification module, every incoming packet is prepended with three mandatory parameters – an *arrival* time, a packet *size* in bytes, and a destination *queue* – and three optional parameters – an intrinsic *value* (whose meaning is application-specific), the *processing* requirement in virtual cycles, and *slack* (maximal offset in time from *arrival* when the packet must be transmitted). We assume that these properties are set by an external *classification unit* (e.g., OpenFlow [24], if a virtual switch is defined with the finest possible resolution), except for *arrival* (which is set by BASEL when a packet is received) and *size*.

Figure 1 shows an abstract representation of the `Packet` data structure. Intrinsic *value* and *processing* requirements can be useful to define prioritization levels [20]. The *slack* represents a time bound, which can be used in management decisions of latency-sensitive applications; for instance, if buffer occupancy already exceeds the *slack* value of an incoming packet, the packet can be dropped during admission even if there is available buffer space. In Section 5, we will see specific examples that exploit these characteristics.

We postulate that all decisions of buffer management policies (during admission or scheduling) are based only on the specified packet parameters and internal state variables of a buffering architecture (e.g., buffer occupancy).

## 4.2 Queues

Figure 2 summarizes the API provided to the programmer to declare queues. The standard property *size* is defined by the user at declaration time. The *currSize* property changes dynamically as the queue changes its size. Abstractly, a queue contains packets ordered according to user-defined priorities for admission control and processing policies. In BASEL, we consider two user-defined priorities:

(a) `procPrio(p1,p2)` is a Boolean function that takes two abstract packets and returns *true* only if *p1* has a higher processing priority than *p2*. We call functions that compare any two objects of the same type *comparators* (defined as Boolean expressions with arithmetic/Boolean operators and access to packet and object attributes), so `procPrio` is a packet comparator. In BASEL we are only concerned with the highest processing priority packet at any point. Hence, the only way to access the queue ordered by `procPrio` is through the `getHOL()` primitive which returns the HOL (i.e., highest processing priority as defined by `procPrio`) packet in the queue. E.g., to encode a FIFO processing priority the user sets `procPrio(p1,p2) = p1.arrival < p2.arrival`. With this definition, each call to `getHOL()` returns the packet in the queue with the oldest arrival time.

(b) `admPrio(p1,p2)` is also a packet comparator used in case of congestion to choose the packets that should be dropped

---

```

Port {
  // primitive properties
  getBestQueue() // on weightSched    [queue fun]
  getCurrQueue() // scheduled one     [queue fun]

  // user-specified at declaration
  schedPrio(q1,q2) // compare q-s     [bool fun]
  postSchedAct() // {MARK,NOTIFY,..} [action fun]
}

```

---

Figure 3: BASEL’s port primitive

---

```

Buffer {
  // primitive properties
  currSize      // current size      [r, dyn]
  getBestQueue() // on weightAdm     [queue fun]
  getCurrQueue() // admitted one     [queue fun]

  // user-specified at declaration
  size          // size              [r, cons]
  congestion()   // cong. predic.     [bool fun]
  queuePrio(q1,q2) // compare q-s    [bool fun]
  postAdmAct()  // {MARK,NOTIFY,..} [action fun]
}

```

---

Figure 4: BASEL’s buffer primitive

from the queue to restore the queue into a decongested state. We could have simply chosen to use the least valuable packets according to `procPrio` for drops, but we will see in Sec. 5 that separate priorities for admission and processing has more flexibility and improves performance.

To indicate when a queue is virtually *congested*, we use a user-defined predicate `congestion()`. The optional function `postAdmAct()` returns an action applied after admission and can update `weightAdm` (if necessary). The function `postAdmAct()` can also be used to implement *explicit congestion notifications* [8] or *backpressure*; `postAdmAct()` can return actions as **MARK**, **NOTIFY**, etc. For cases when bandwidth is allocated not only with respect to packet attributes, queues maintain a `weightSched` variable that can be updated dynamically after each scheduling operation. With `weightSched` one can, e.g., define static bandwidth allocation among queues of the same port during scheduling decisions; `weightSched` can be updated in the `postSchedAct()` function which is defined at the port level.

### 4.3 Ports

The interface provided for ports is presented in Figure 3. Each port manages a set of queues assigned to the port at its declaration.<sup>2</sup> The policy `schedPrio(q1,q2)` is a user-defined (queue comparator) scheduling property that defines which HOL packet is scheduled next (this packet is accessed through the `getBestQueue()` function). For example, a priority based on packet values which implements several levels of strict priorities is declared as follows:

```

schedPrio(q1,q2) =
  q1.getHOL().value > q2.getHOL().value

```

Finally, `postSchedAct()` is similar to the `postAdmAct()` function of queues which can be used to define new services.

### 4.4 Buffers

The interface provided for declaring buffers is presented in Figure 4. A buffer is an optional entity; it is declared only in the case when several queues share buffer space. Each buffer manages a set of queues assigned to it at creation time; `congestion()`, `postAdmAct()`, `size`, and `currSize` are similar to the corresponding queue attributes. In case of congestion, an admission control policy on the buffer level finds a queue whose packet should be dropped, and the admission control policy of the chosen

<sup>2</sup>We leave the `new` operator used to create network objects in BASEL implicit; its usage will be clear from the examples in Sec. 5.

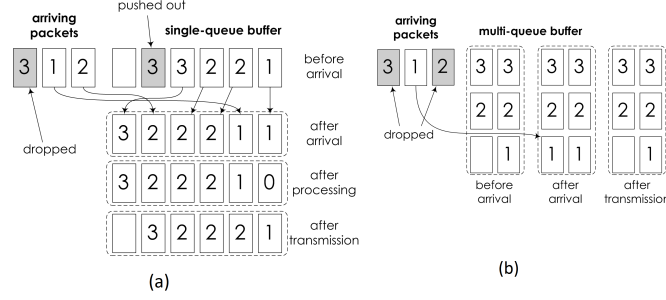


Figure 5: Left: single priority queue with buffer  $B = 6$ ; right: multi-queued switch with three queues ( $k = 3$ ) and buffer  $B = 2$  each. Dashed lines enclose queues.

admPrio	procPrio	OPT/ALG
fifo()	fifo()	$O(k)$
fifo()	srpt()	$O(\log k)$
rsrpt()	srpt()	1 (optimal)

Table 1: Sample BASEL policies with analytic results, single queue architecture;  $k$  is the maximal processing requirement, OPT/ALG is the competitive ratio.

queue determines which packet is dropped. To order queues for admission, the user specifies the `queuePrio` comparator. For instance, to implement LQD the following comparator can be used:

```
queuePrio(q1,q2) = q1.currSize < q2.currSize
```

## 5 BASEL at Work (examples)

To demonstrate the impact of admission and processing orders on the efficiency of admission control, consider throughput maximization in a single queue buffering architecture (buffer of size  $B$ ), each unit-sized and unit-valued packet assigned with a number of required processing cycles ranging from 1 to  $k$  (see Figure 5(a)). Figure 6 shows a sample definition in BASEL for a single queue buffering architecture, where we deploy a single `q1` in a single port `out`. Below we will see how to define different management policies in this architecture. In this example, the `fifo()` packet comparator uses arrival time, the `srpt()` (shortest remaining processing time) and `rsrpt()` (reversed shortest remaining processing time) comparators use remaining processing requirements. The congestion condition declared in `defCongestion()` is trivial, satisfied when occupancy exceeds queue size. We instantiate a single queue `q1` with this congestion policy.

```
// considered priorities for admission and
// processing
fifo(p1,p2) = (p1.arrival < p2.arrival)
srpt(p1,p2) = (p1.processing < p2.processing)
rsrpt(p1,p2) = (p1.processing > p2.processing)

// default congestion condition for all
// considered policies
defCongestion() = lambda q, (q.currSize >= q.size)

// initializing a generic buffering architecture
q1=Queue(B); out=Port(q1);
q1.proPrio(p1,p2)=fifo(p1,p2);
q1.congestion=defCongestion(q1);
```

Figure 6: Single queue buffering architecture in BASEL

Table 1 lists implementations for `admPrio` and `procPrio` in this architecture and analytic competitiveness results for various online policies versus the optimal offline algorithm [20, 28]. Each row represents a management algorithm for a single queue;

e.g., the first row shows a simple greedy algorithm that admits every incoming packet if possible (see `congestion()`), and processes them in `fifo()` order; it is  $O(k)$ -competitive for maximum processing requirement  $k$ . In BASEL, this algorithm looks as follows:

```
q1.admPrio=fifo; q1.procPrio=fifo;
```

On the other hand, changing the processing order `fifo()` to `srpt()` introduces a significant improvement in performance and this version of the greedy policy is already  $O(\log(k))$ -competitive. With the third greedy algorithm that processes packets in `srpt()` order and admits them in `rsrpt()` order, we get an optimal algorithm for throughput maximization regardless of traffic distributions [20]. Since here a port manages only one queue, a *scheduling policy* is just an implicit call to `getHOL()`.

One alternative architecture for packets with heterogeneous processing requirements is to allocate queues for packets with the same processing requirements (see Figure 5(b)). The following code creates this buffering architecture in BASEL, where  $k$  queues share an equal portion of memory  $B/k$ .

```
// creation of buffering architecture
q1=Queue(B/k);...qk=Queue(B/k);
out=Port(q1,...,qk);
```

In this architecture, there is no need for advanced processing and admission orders since only packets with the same processing requirement are admitted in the same queue. The following BASEL code instantiates `admPrio`, `procPrio` and `congestion` in the  $k$  created queues.

```
q1.admPrio=fifo; ...; qk.admPrio=fifo;
q1.procPrio=fifo; ...; qk.procPrio=fifo;
q1.congestion=defCongestion(q1); ...;
qk.congestion=defCongestion(qk);
```

This change of buffering architecture is not for free since the buffer of these queues is not shareable. But even here, the decision of which packet should be processed to maximize throughput is non-trivial since it is unclear which characteristic is most relevant for throughput optimization: buffer occupancy, required processing, or a combination. BASEL code in Figure 7 presents six different scheduling priorities and `postSchedAct` actions in the cases when this action is used.

---

```
// LQF: HOL packet from Longest-Queue-First
lqf(q1,q2) = (q1.currSize > q2.currSize);
// SQF: HOL packet from Shortest-Queue-First
sqf(q1,q2) = (q1.currSize < q2.currSize);
// MAXQF: HOL packet from queue that
// admits max processing
maxqf(q1,q2)= (q1.weightSched > q2.weightSched);
// MINQF: HOL packet from queue that admits
// min processing
minqf(q1,q2)= (q1.weightSched < q2.weightSched);
// CRR: Round-Robin with per cycle resolution
crr(q1,q2) = (q1.weightSched < q2.weightSched);
crrPostSchedAct() = lambda port,
    (port.getCurrQueue().weightSched += k);
// PRR: Round-Robin with per packet resolution
prr(q1,q2) = (q1.weightSched < q2.weightSched);
prrPostSchedAct() = lambda port,
    (let q = port.getCurrQueue() in
        if (q.getHOL().processing == 0)
            q.weightSched += k*k));
```

---

Figure 7: BASEL example of scheduling priorities and `postSchedAct` actions for multiple separated queues.

Table 2 summarizes various online scheduling policies as shown in [21, 28]. Observe that buffer occupancy is not a good characteristic for throughput maximization: `lqf()` and `sqf()` have bad competitive ratios, while a simple greedy scheduling policy Min-Queue-First (MQF) that processes the HOL packet from the non-empty queue with minimal required processing (`minqf()`) is 2-competitive. This means that MQF will have optimal throughput with a moderate speedup of 2 [21]. The other two policies that implement fairness with per-cycle or per-packet resolution (CRR and PRR respectively) have relatively weak performance; this demonstrates the fundamental tradeoff between fairness and throughput. The following code snippet in BASEL, for instance, corresponds to the CRR policy:

init. weightSched	postSchedAct	schedPrio	OPT/ALG
unused	unused	lqf ()	$\Omega(\frac{B}{2})$
unused	unused	sqf ()	$\Omega(k)$
unused	unused	maxqf ()	$\Omega(k)$
qi.weightSched=i	unused	minqf ()	upper bound 2
qi.weightSched=i	crrPostSchedAct ()	crr ()	$\Omega(\frac{k}{\ln k})$
qi.weightSched=i	prrPostSchedAct ()	prr ()	$\Omega(\frac{3k(k+2)}{4k+16})$

Table 2: Examples of policies in BASEL for multiple queues architecture with analytic results;  $k$  is a maximal processing requirements,  $B$  is a buffer size of a single queue. OPT/ALG is a throughput of an optimal offline OPT algorithm vs. online algorithm ALG.

```
// initializing schedWeight for CRR
q1.weightSched=1; ... qk.weightSched=k;
// initial. postSchedAct to update schedWeight
out.postSchedAct = crrPostSchedAct(out);
```

Currently, the best tools available to evaluate performance of buffering architectures are discrete simulators such as NS-2 [6] or OMNet++ [4] that can use traffic traces and/or various traffic distributions to analyze performance of buffering architectures by specifying management policies in a high level language. Due to its simplicity, BASEL can be used as a discrete simulator whose configuration is limited to several user-defined expressions. For instance, Figures 9 and 8 show the impact of admission, processing, and scheduling policies on throughput optimization for a single queue and multiple queues buffering architectures with packets of heterogeneous processing requirements; in these examples, traffic was generated with an ON-OFF Markov modulated Poisson process (MMPP) with Poisson arrival processes with intensity  $\lambda$ , and required processing chosen uniformly at random from  $1..k$ . But even if we know how to represent arrivals and analyze them, the applicability of these results will be limited to specific settings. Hence, BASEL is being developed for deployment on real systems.

## 6 BASEL Internals

One of the fundamental building blocks in BASEL is the *priority queue* data structure where the order of elements is based on a user-defined priority. The implementation keeps a single copy of the packets, and priority queues are implemented with pointers to actual packets. Since a virtual buffering architecture can be defined with switch resolution (or a part of it), BASEL implementation is reduced to efficient implementation of a priority queue data structure that can operate at line rate. While in the general case priority queue operations take  $O(\log(N))$  time, where  $N$  is its size, there are restricted versions (e.g., for a predefined range of priorities) that can support most operations in  $O(1)$  and can be efficiently implemented even in hardware [18, 26]. To guarantee a constant number of insert/remove and lookup operations during admission or scheduling of a packet (i.e., to avoid rebuilding the priority queue), user-defined expressions for priorities are fixed during operation.

A *push-out mechanism* makes an architecture capable to push out already admitted packets. This mechanism is supported in BASEL. To avoid different implementations for the push-out and non-push-out cases, an admission control policy always virtually admits an incoming packet. In the event of a virtual congestion, admission control drops the least valuable packets until congestion is lifted. The complexity of BASEL is reduced to translating user-defined settings to a target system that implements a virtual buffering architecture. In some cases, the target should be extended, or the expressiveness of BASEL can be restricted. We are currently implementing BASEL on top of the Open vSwitch (OVS) as a sample target architecture with the finest resolution [5, 29].

### 6.1 BASEL implementation in Open vSwitch

OVS implements the control plane in user space and the data plane in the kernel. To support BASEL on the control plane, we extend OVSDb (Open vSwitch Database) with the notions of port, queue, and buffer described in Section 4. Moreover, since OVS exploits Linux TC (Traffic Control) kernel modules via the `netdev-linux` library to manipulate queuing and scheduling disciplines (`qdisc`), we are also adding configuration options to TC to express BASEL's admission, processing, and scheduling policies. Similar extensions are being added on the data plane via Linux kernel TC loadable kernel modules.

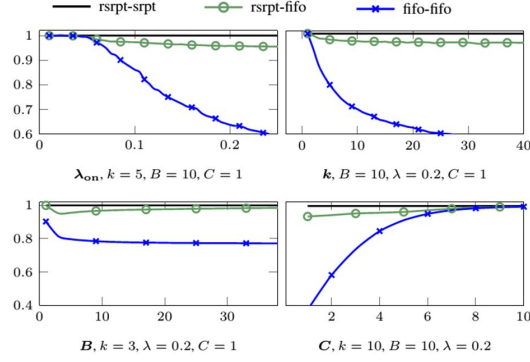


Figure 8: Optimal vs three online algorithms for a single queue architecture with heterogeneous processing;  $y$ -axis, competitive ratio;  $x$ -axis, top to bottom, left to right:  $\lambda$ ; max required processing  $k$ ; buffer size  $B$ ; speedup  $C$ .

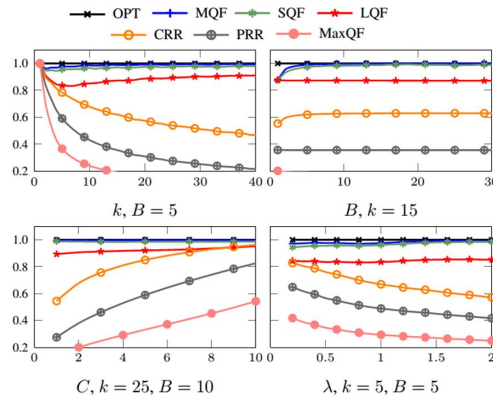


Figure 9: Online vs optimal algorithms for multiple queues with heterogeneous processing;  $y$ -axis, competitive ratios;  $x$ -axis, top to bottom, left to right: max required processing  $k$ , buffer size  $B$ , speedup  $C$ , intensity  $\lambda$ .



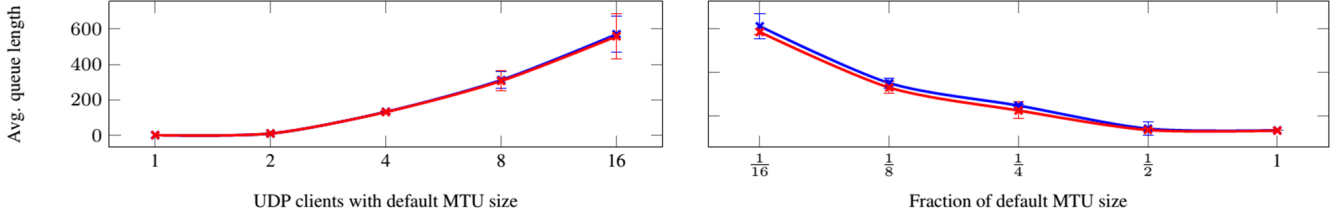
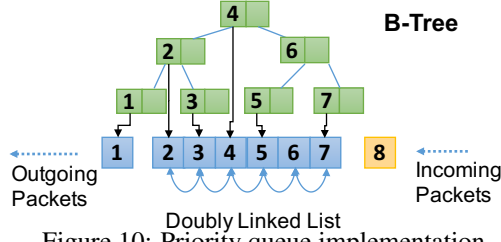


Figure 11: *Left*: average queue length as a function of number of clients generating UDP traffic with default MTU size. *Right*: fraction of default MTU size; blue: FIFO with prioritization; red: regular FIFO.

## 6.2 BASEL feasibility

We have extended Linux’s default `qdisc`<sup>3</sup> (i.e., `pfifo_fast`) to support packet prioritization based on arrival time. Instead of modifying the underlying default packet queue (a doubly linked list), we use an existing B-Tree implementation on top of a default FIFO queue to manage packet prioritization to preserve backward compatibility to existing `qdisc` solutions. As shown on Figure 10, we add a reference to the enqueueing packets to the B-Tree and the highest priority packet (i.e., the earliest arrival time) is dequeued first. We remark that FIFO does not need to utilize a B-Tree in general; we use it as a baseline to explore the performance overhead of a generic implementation of prioritization.

In our testbed, we set a 3-node line topology to measure the performance overhead of our packet prioritization logic. Figure ?? shows that the middle node runs OVS with modified data plane (Linux kernel) and acts as a pass-through switch. We vary the number of parallel traffic generators on the first node and measure average queue length (i.e., number of packets in the default queue) in a receiver node on the third for two `qdisc`s: default FIFO and extended FIFO with prioritization, reporting the average value of 50 runs with 95% confidence interval. Figure 11(left) shows the average queue lengths for the two `qdisc`s; in both cases, average queue length increases with the number of UDP clients. In FIFO with 16 clients, the most congested case, regular FIFO has average queue length 559.333 vs. 571 for FIFO with prioritization, only a 2% degradation. We also varied MTU sizes in the same 3-node line topology testbed with 4 parallel UDP generators, which is a good enough case to observe queue build-ups but not dropping packets in the pass-through switch. We measured average queue lengths of the two `qdisc`s by varying MTU sizes from  $\frac{1}{16}$  of the default MTU size to its default size (1500 bytes). Figure 11(right) shows that for both `qdisc`s the average queue length decreases as MTU size increases; FIFO with prioritization incurs only 4% overhead: for MTU size of  $\frac{1500}{16}$  bytes the result is 584.3 vs. 610.7. Hence, we conclude that packet prioritization on top of FIFO incurs negligible performance overhead.

## 7 Conclusion

We propose a simple yet expressive language to define buffering architectures and their management policies. The proposed language is independent from a classification module and can define buffering architectures and their management policies with any resolution from a single network element to a virtual switch that can represent the whole network or a part of it. We believe that the efficient representation of buffering architectures in BASEL can enable and accelerate innovation in this domain.

<sup>3</sup>Queuing Discipline (`qdisc`) is an integral part of Linux Traffic Controlling (TC) used to shape outgoing (egress) traffic for an interface; `qdisc` has an enqueue method to handle outgoing packets and a dequeue method to fetch packets written to the network interface.

## References

- [1] Cisco 12000 series modular gigabit ethernet line card. <http://www.cisco.com/c/en/us/products/interfaces-modules/12000-series-modular-gigabit-ethernet-line-card/index.html>.
- [2] The cisco quantumflow processor: Cisco's next generation network processor. [http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution\\_overview\\_c22-448936.html](http://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html).
- [3] <http://www.cisco.com/networkers/nw04/presos/docs/rst-4314.pdf>. <http://www.cisco.com/networkers/nw04/presos/docs/RST-4314.pdf>.
- [4] OMNeT++. <http://www.omnetpp.org/>.
- [5] Open vSwitch. <http://www.openvswitch.org>.
- [6] This is the ns-2 wiki. [http://nsnam.isi.edu/nsnam/index.php/Main\\_Page](http://nsnam.isi.edu/nsnam/index.php/Main_Page).
- [7] W. Aiello, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. *ACM Trans. on Algorithms*, 5(1), 2008.
- [8] S. Bauer, R. Beverly, and A. Berger. Measuring the state of ECN readiness in servers, clients, and routers. In *IMC*, pages 171–180, 2011.
- [9] BBC News. US Watchdog to Propose New Net Neutrality Rules, 2014. <http://www.bbc.com/news/technology-27141121>.
- [10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *CCR*, 44(3):87–95, 2014.
- [11] A. J. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, pages 1–12, 1989.
- [12] W. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. Netw.*, 10(4):513–528, 2002.
- [13] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: an API for application control of sdns. In *SIGCOMM*, pages 327–338, 2013.
- [14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.
- [15] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *ICFP*, pages 279–291, 2011.
- [16] J. Gettys. Low latency requires smart queuing:traditional AQM is not enough!, 2013. [http://www.internetsociety.org/sites/default/files/pdf/accepted/29\\_bis\\_ISOC\\_Workshop\\_2.pdf](http://www.internetsociety.org/sites/default/files/pdf/accepted/29_bis_ISOC_Workshop_2.pdf).
- [17] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, pages 15–26, 2013.
- [18] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw.*, 15(2):450–461, 2007.
- [19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined wan. In *SIGCOMM*, pages 3–14, 2013.
- [20] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.
- [21] K. Kogan, A. López-Ortiz, S. I. Nikolenko, and A. Sirotkin. Multi-queued network processors for packets with heterogeneous processing requirements. In *COMSNETS*, pages 1–10, 2013.

- [22] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *INFOCOM*, pages 830–838, 2010.
- [23] P. E. McKenney. Stochastic fairness queueing. In *INFOCOM*, pages 733–740, 1990.
- [24] N. McKeown, G. Parulkar, S. Shenker, T. Anderson, L. Peterson, J. Turner, H. Balakrishnan, and J. Rexford. OpenFlow switch specification, 2011. <http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf>.
- [25] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. In *NSDI*, pages 1–13, 2013.
- [26] A. Morton, J. Liu, and I. Song. Efficient priority-queue data structure for hardware implementation. In *FPL*, pages 476–479, 2007.
- [27] K. M. Nichols and V. Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, 2012.
- [28] S. I. Nikolenko and K. Kogan. Single and multiple buffer processing. In *Encyclopedia of Algorithms*. Springer, 2015.
- [29] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *USENIX NSDI*, pages 117–130, 2015.
- [30] A. Shieh, S. Kandula, A. G. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.
- [31] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No silver bullet: extending SDN to the data plane. In *HotNets*, pages 19:1–19:7, 2013.
- [32] H. Song. Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane. In *HotSDN*, pages 127–132, 2013.
- [33] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. D. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, pages 213–226, 2014.
- [34] D. L. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–17, 1996.
- [35] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: simplifying SDN programming using algorithmic policies. In *SIGCOMM*, pages 87–98, 2013.